МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и Структуры Данных»

Тема: БДП с рандомизацией.

Студент гр. 8304 _____ Самакаев Д.И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент: Самакаев Дмитрий Ильич

Группа 8304

Тема работы: БДП с рандомищацией.

Исходные данные:

По заданному файлу F (типа file of Elem), все элементы которого различны, построить структуру данных определённого типа – БДП или хеш-таблицу;

Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если не входит, то добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Содержание пояснительной
запик

- Содержание
- Введение
- Тестирование
- Исходный код
- Использованные
  источники

Дата выдачи задания: 11.10.2019

Дата сдачи реферата: 25.12.2019

Дата защиты реферата:

25.12.2019

Студент _____ Самакаев Д.И.

Преподаватель _____ Фирсов

_____

_____ M.A.

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++, которая сочетает в себе несколько функций: кодирования/декодирования текста. Были использованы преимущества C++ для минимизации кода. Для лучшего понимания кода было в нем приведено большое кол-во отладочных выводов. Также была проведена его оптимизация с целью экономии выделяемой в процессе работы памяти и улучшения быстродействия программы.

SUMMARY

In this work, a program was created in the C ++ programming language, which combines several functions: encoding / decoding text. The benefits of C ++ were used to minimize code. For a better understanding of the code, it contained a large number of debugging outputs. Also, its optimization was carried out in order to save the memory allocated in the process of working and improve the speed of the program.

СОДЕРЖАНИЕ

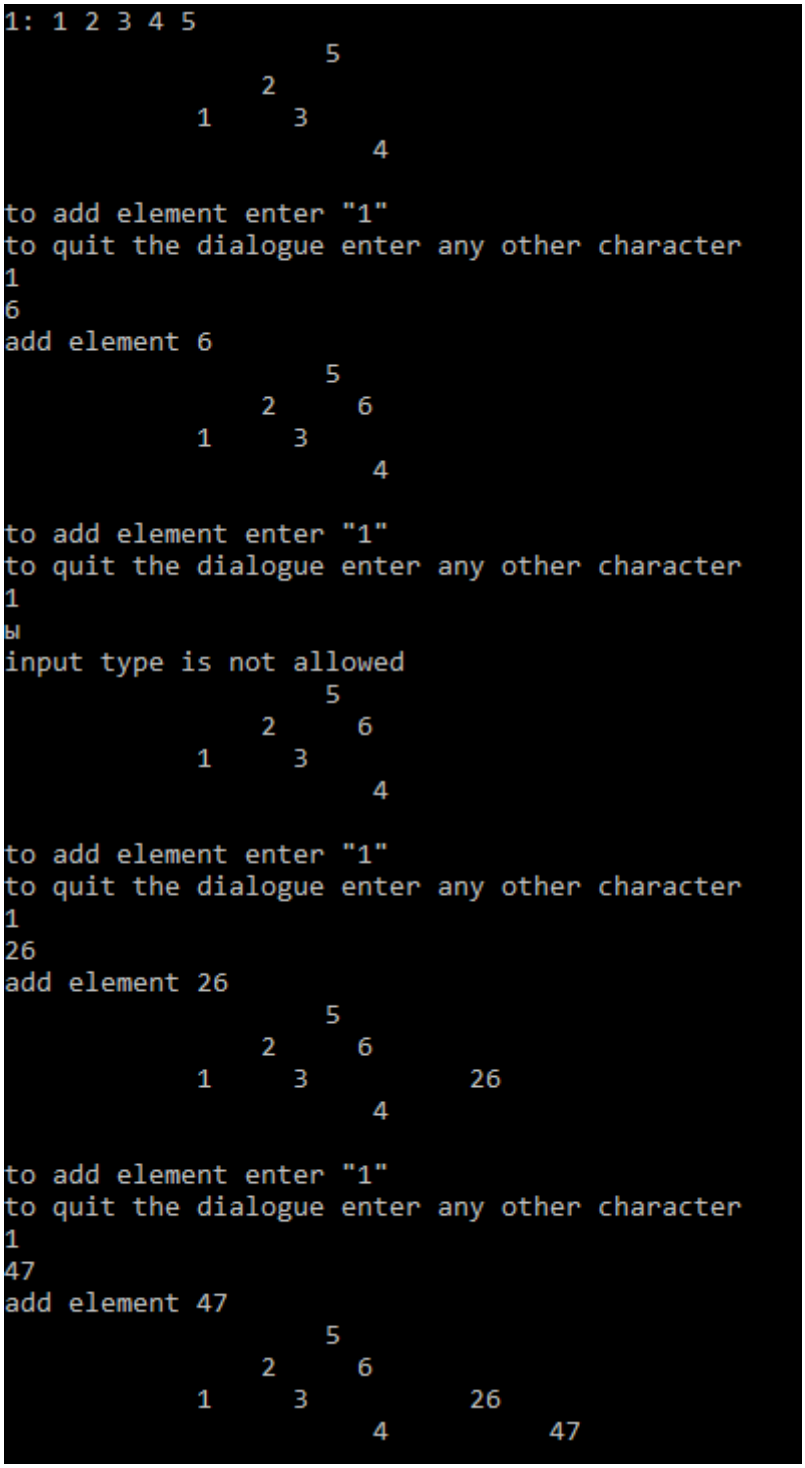
Введение

Целью данной курсовой работы является реализация БДП и демонстрация работы алгоритма. Алгоритм использует рекурсивные методы. Двоичное дерево поиска (англ. binary search tree, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X

На рисунке 1 продемонстрирована работа программы.

Было реализовано общение с пользователем посредством консольного интерфейса.



(Рисунок 1. Ход работы программы)

Из тестового файла вводится строка с входными данными. На их основе строится дерево.

1. ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

1 Структуры:

```
template<typename T>
struct Branch {
   Branch() = default;

   std::shared_ptr<T> root = nullptr;
   std::shared_ptr<Branch> left = nullptr;
   std::shared_ptr<Branch> right = nullptr;


};
```

Структура Branch представляет узел дерева, где left –левая ветка, right –правая, a root – указатель на сам элемент.

```
template<typename T>
class BinTree {
public:
	BinTree() {
		head = std::make_shared<Branch<T>>();
	}

	bool unit_insist(T unit, std::shared_ptr<Branch<T>> temp) {
		if (!temp->root) {
			return 0;
		}

		if (*(temp->root) > unit) {
			if (!temp->left)
				return 0;
			return unit_insist(unit, temp->left);
		}

		else if (*(temp->root) < unit) {
			if (!temp->right)
				return 0;
			return unit_insist(unit, temp->right);
		}

		return 1;
	}

	//returns TRUE if tree already has unit and FALSE if it has not
	bool Random_add_root(T unit, std::shared_ptr<Branch<T>> temp) {

		if (!unit_insist(unit, head)) {
			size_t random_tree_number = rand() % (size + 2);

			if (random_tree_number == size + 1) {

				if (!temp->root) {
					temp->root = std::make_shared<T>();
```

```
                        *(temp->root) = unit;
                        size++;
                        return false;
                  }

                  if (*(temp->root) > unit) {
                        std::shared_ptr<Branch<T>> new_head;

                        new_head = std::make_shared<Branch<T>>();
                        new_head->right = head;
                        new_head->root = std::make_shared<T>();
                        *(new_head->root) = unit;

                        head = new_head;
                        size++;
                  }
                  else {
                        std::shared_ptr<Branch<T>> new_head;

                        new_head = std::make_shared<Branch<T>>();
                        new_head->left = head;
                        new_head->root = std::make_shared<T>();
                        *(new_head->root) = unit;

                        head = new_head;
                        size++;
                  }
            }
            else addRoot(unit, temp);

            return true;
      }
      else
            return false;

}

bool addRoot(T unit, std::shared_ptr<Branch<T>> temp) {

      if (!temp->root) {
            temp->root = std::make_shared<T>();
            *(temp->root) = unit;
            size++;
            return false;
      }

      if (*(temp->root) > unit) {
            if (!temp->left)
                  temp->left = std::make_shared<Branch<T>>();
            return addRoot(unit, temp->left);
      }

      else if (*(temp->root) < unit) {
            if (!temp->right)
                  temp->right = std::make_shared<Branch<T>>();
            return addRoot(unit, temp->right);
      }
```

```
            return true;
      }

      void fill_map(std::shared_ptr<Branch<T>> temporary, size_t depth,
std::map<size_t, std::string>& depth_root_map) {

            depth++;

            if (temporary->left)
                  fill_map(temporary->left, depth, depth_root_map);
            else depth_root_map[depth + 1] += "      ";
            if(temporary->right)
                  fill_map(temporary->right, depth, depth_root_map);
            else depth_root_map[depth + 1] += "      ";

            if ((depth_root_map.find(depth) != depth_root_map.end()) &&
(temporary->root))
                  depth_root_map[depth] += std::to_string(*(temporary->root)) + "
";
            else if (temporary->root) {
                  depth_root_map.insert(make_pair(depth,
std::to_string(*(temporary->root))));
                  depth_root_map[depth] += "      ";
            }
      }


      void beauty_print_tree(std::map<size_t, std::string>& depth_root_map) {
            size_t j = 0;
            for (auto it = depth_root_map.begin(); it != depth_root_map.end();
it++) {
                  for (int i = 0; i < depth_root_map.size() - j; i++)
                        std::cout << "    ";
                  std::cout << it->second << std::endl;
                  j++;
            }
      }

      //возвращает количество успешно добавленных элементов
      void fill_tree(std::vector<T> numbers) {

            for (size_t i = 0; i < numbers.size(); i++)
                  Random_add_root(numbers.at(i), head);
            size++;
      }

      void task(T unit) {
            if (Random_add_root(unit, head))
                  std::cout << "add element " << unit << std::endl;
            else std::cout << "stucture already has element "<< unit <<
std::endl;
      }

      void print_tree() {

            fill_map(head, 0, depth_root_map);
```

```
            beauty_print_tree(depth_root_map);
            depth_root_map.clear();
      }

      std::shared_ptr<Branch<T>>getHead() {
            return head;
      }

private:
      std::shared_ptr<Branch<T>> head;
      size_t size = 0;
      std::map<size_t, std::string> depth_root_map;
  };
```

Структура bin_tree нужна для хранения и обработки дерева.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была написана программа, содержащая в себе реализацию БДП. Был получен опыт работы с дополнительными возможностями C++ и изучены алгоритмы работы с БДП. Также были закреплены знания полученные на протяжении семестра. Исходный код программы находится в приложении А.

ПРИЛОЖЕНИЕ А

СОДЕРЖИМОЕ ФАЙЛА CW.CPP

```
#include "lab.h"


template<typename T>
bool str_to_t(T& item) {
	std::string str;
	std::cin >> str;

	std::stringstream linestream(str);
	linestream >> item;

	char c;
	if ((!linestream) || (linestream >> c)) {
		std::cout << "input type is not allowed" << std::endl;
		return false;
	}

	return true;
}

template<typename T>
void dialogue(BinTree<T> tree) {

	char option;
	T unit;
	tree.print_tree();
	std::cout << "to add element enter \"1\"\nto quit the dialogue enter any other
character\n";

	option = getchar();
	getchar();
	switch (option){
	case '1':
		if(str_to_t(unit))
			tree.task(unit);
		getchar();
		dialogue(tree);
		break;
	default:
		break;
	}
}

template<typename T>
void file_input(char* argv) {

	std::ifstream file;
	std::string testfile = argv;

	file.open(testfile);
	if (!file.is_open()) {
		std::cout << "Error! File isn't open" << std::endl;
		return;
	}

	size_t i = 1;
```

```
	std::string expression;

	while (getline(file, expression)) {
		std::vector<T> units;
		std::istringstream iss(expression);

		BinTree<T> tree;
		T unit;
		while (iss >> unit) {
			units.push_back(unit);
		}

		tree.fill_tree(units);

		std::cout << i << ": " << expression << std::endl;
		i++;

		dialogue<T>(tree);
	}

}

int main(int argc, char** argv) {

	if (argc == 1) {
		std::cout<<"Please check you entered test file name";
	}
	else file_input<int>(argv[1]);
   }
```

СОДЕРЖИМОЕ ФАЙЛА BIN_TREE.H

```
#pragma once
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <vector>
#include <sstream>
#include <cstdlib>
#include <map>


template<typename T>
struct Branch {
	Branch() = default;

	std::shared_ptr<T> root = nullptr;
	std::shared_ptr<Branch> left = nullptr;
	std::shared_ptr<Branch> right = nullptr;

};

template<typename T>
class BinTree {
public:
	BinTree() {
		head = std::make_shared<Branch<T>>();
	}
```

```
bool unit_insist(T unit, std::shared_ptr<Branch<T>> temp) {
	if (!temp->root) {
		return 0;
	}

	if (*(temp->root) > unit) {
		if (!temp->left)
			return 0;
		return unit_insist(unit, temp->left);
	}

	else if (*(temp->root) < unit) {
		if (!temp->right)
			return 0;
		return unit_insist(unit, temp->right);
	}

	return 1;
}

//returns TRUE if tree already has unit and FALSE if it has not
bool Random_add_root(T unit, std::shared_ptr<Branch<T>> temp) {

	if (!unit_insist(unit, head)) {
		size_t random_tree_number = rand() % (size + 2);

		if (random_tree_number == size + 1) {

			if (!temp->root) {
				temp->root = std::make_shared<T>();
				*(temp->root) = unit;
				size++;
				return false;
			}

			if (*(temp->root) > unit) {
				std::shared_ptr<Branch<T>> new_head;

				new_head = std::make_shared<Branch<T>>();
				new_head->right = head;
				new_head->root = std::make_shared<T>();
				*(new_head->root) = unit;

				head = new_head;
				size++;
			}
			else {
				std::shared_ptr<Branch<T>> new_head;

				new_head = std::make_shared<Branch<T>>();
				new_head->left = head;
				new_head->root = std::make_shared<T>();
				*(new_head->root) = unit;

				head = new_head;
				size++;
			}
		}
		else addRoot(unit, temp);
```

```
                return true;
        }
        else
                return false;

}

bool addRoot(T unit, std::shared_ptr<Branch<T>> temp) {

        if (!temp->root) {
                temp->root = std::make_shared<T>();
                *(temp->root) = unit;
                size++;
                return false;
        }

        if (*(temp->root) > unit) {
                if (!temp->left)
                        temp->left = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->left);
        }

        else if (*(temp->root) < unit) {
                if (!temp->right)
                        temp->right = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->right);
        }

        return true;
}

void fill_map(std::shared_ptr<Branch<T>> temporary, size_t depth, std::map<size_t,
std::string>& depth_root_map) {

        depth++;

        if (temporary->left)
                fill_map(temporary->left, depth, depth_root_map);
        else depth_root_map[depth + 1] += "     ";
        if(temporary->right)
                fill_map(temporary->right, depth, depth_root_map);
        else depth_root_map[depth + 1] += "     ";

        if ((depth_root_map.find(depth) != depth_root_map.end()) && (temporary->root))
                depth_root_map[depth] += std::to_string(*(temporary->root)) + "     ";
        else if (temporary->root) {
                depth_root_map.insert(make_pair(depth, std::to_string(*(temporary-
>root))));
                depth_root_map[depth] += "     ";
        }
}


void beauty_print_tree(std::map<size_t, std::string>& depth_root_map) {
        size_t j = 0;
        for (auto it = depth_root_map.begin(); it != depth_root_map.end(); it++) {
                for (int i = 0; i < depth_root_map.size() - j; i++)
                        std::cout << "    ";
                std::cout << it->second << std::endl;
```

```
                j++;
            }
        }

        //возвращает количество успешно добавленных элементов
        void fill_tree(std::vector<T> numbers) {

            for (size_t i = 0; i < numbers.size(); i++)
                Random_add_root(numbers.at(i), head);
            size++;
        }

        void task(T unit) {
            if (Random_add_root(unit, head))
                std::cout << "add element " << unit << std::endl;
            else std::cout << "stucture already has element "<< unit << std::endl;
        }

        void print_tree() {

            fill_map(head, 0, depth_root_map);
            beauty_print_tree(depth_root_map);
            depth_root_map.clear();
        }

        std::shared_ptr<Branch<T>>getHead() {
            return head;
        }

    private:
        std::shared_ptr<Branch<T>> head;
        size_t size = 0;
        std::map<size_t, std::string> depth_root_map;

    };
```